CMPUT 229 - Computer Organization and Architecture I
Final Exam — Fall 2003 - Section A1

Prof. José Nelson Amaral
Computing Science Department
University of Alberta

Name:

CMPUT 229 Honor Code

By turning in the exam solution for grading, I certify that I have worked all the solutions on my own, that I have not copied or transcribed solutions from a classmate, someone outside the class, or from any other source. I also certify that I have not facilitated or allowed any of my classmates to copy my own solutions. I am aware that the violation of this honor code constitutes a breach of the trust granted me by the teaching staff, compromises my reputation, and subjects me to the penalties prescribed in Section 26.1 of the University of Alberta 2003/2004 Calendar.


<table>
<thead>
<tr>
<th>Question</th>
<th>/30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question 1</td>
<td>/30</td>
</tr>
<tr>
<td>Question 2</td>
<td>/20</td>
</tr>
<tr>
<td>Question 3</td>
<td>/20</td>
</tr>
<tr>
<td>Question 4</td>
<td>/30</td>
</tr>
<tr>
<td>Total</td>
<td>/100</td>
</tr>
<tr>
<td>Curving</td>
<td>/100</td>
</tr>
<tr>
<td>Rank</td>
<td>/100</td>
</tr>
</tbody>
</table>

1
Question 1 (30 points):

The function tak is a recursive function created by Ikuo Takeuchi that does not do anything useful, but is a good example of a function that executes many recursive calls. The C code for tak is shown below.

```c
int tak(int x, int y, int z)
{
    int temp;
    if(y < x)
    {
        temp = tak(tak(x-1, y, z), tak(y-1, z, x), tak(z-1, x, y));
        return(1 + temp);
    }
    else return z;
}
```

When solving this question, follow the MIPS register saving convention:

Registers $t0$–$t9$ are caller-saved: they are not preserved by the callee on a procedure call;

Registers $s0$–$s7$ are callee-saved: they must be preserved by the callee on a procedure call.

Write the MIPS assembly code for the function tak in Table 1. You will be evaluated on the following items:

a. (10 points) Correct stack manipulation, save all registers that must be saved for the correct execution of the function, and does not save any register that does not need to be saved.

b. (10 points) Correct parameter passing for the recursive calls and correct return of values.

c. (10 points) The entire code works without any error, and is properly commented.
Table 1: Assembly code for tak.
<table>
<thead>
<tr>
<th>Label</th>
<th>Binary Repr.</th>
<th>Hexadecimal</th>
<th>Description(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>1010 0011 0101 1110 1001 0111 1011 1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>1011 1110 0010 1000 0000 0000 0000 0000</td>
<td>0xC114 0000</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>0000 0000 0000 0000 0000 0000 0000 0010</td>
<td>0x1620 0064</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>1100 0001 0001 0100 0000 0000 1111 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6)</td>
<td>0001 0110 0010 0000 0000 0000 1100 1000</td>
<td>0xFFFF FFFD</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Table of binary numbers.

**Question 2 (20 points):** Table 2 contains one 32-bit number in each row. Each row is labeled from (0) to (9).

- (10 points) Complete the second and third columns of the table by writing either the binary or the hexadecimal representation for each row.
- (10 points) Some of the binary representations shown in Table 2 correspond to the descriptions below. Write in the Description column the label corresponding to the descriptions. Some descriptions may not have binary representations in the table. Some binary representations may be described by more than one of the descriptions below. Some binary representations may not correspond to any of the descriptions below.
  
a. Is the number -9.25 represented in IEEE standard floating point representation.

b. Is the value written into $s0$ when the instruction `add $s0, $s0, $s1` is executed. Assume that before this instruction was executed, $s0$ contained the value in row (3) from Table 2 and $s1$ contained the value in row (4) from the same Table.

c. Is the binary representation of the instruction `bne $s1, $zero, label` with offset equal 10010. Remember that R17 = $s1 and R0 = $zero and that the format for the `bne` instruction is as follows.

<table>
<thead>
<tr>
<th>31</th>
<th>Opcode</th>
<th>26</th>
<th>25</th>
<th>rs</th>
<th>21</th>
<th>20</th>
<th>rt</th>
<th>16</th>
<th>15</th>
<th>offset</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

d. Is the value written into $s0$ when the instruction `andi $s0, $s0, 120` is executed. Assume that before this instruction was executed, $s0$ contained the value in row (1) from Table 2.

e. Is the binary representation of the integer -1.

Remember that:

\[ N = (-1)^S \times 1.fraction \times 2^{exponent-127}, 1 \leq exponent \leq 254 \]
\[ N = (-1)^S \times 0.\text{fraction} \times 2^{\text{exponent}-126}, \text{exponent} = 0 \]
Question 3 (20 points): In a program that produces the grade reports for CMPUT 229, ordered is an array of integers containing student ids: ordered[0] contains the id of the student with the highest grade, ordered[1] contains the id of the student with the second highest grade, and so on.

The array names is an array of pointers to null terminated strings containing each student’s name. names is indexed by the student id. The array grades is an array of floats that is also indexed by the student id. Therefore to print a report that shows the students ranked according to their final grade, this program uses the following loop:

```c
printf("Rank\t Name\t Grade\n\n");
for(i = 0 ; i < MAXSTUDENTS ; i++)
    printf("%d\t %s\t %f\n", i, names[ordered[i]], grades[ordered[i]]);
```

The first parameter for the printf function call is a string created by the compiler. Assuming that the value of i is stored in $t0$, write the portion of assembly code necessary to pass the other parameters to printf. Because this printf function is executed inside a loop, you should write a code that executes a minimum number of instructions, and produces the correct output.
**Question 4 (30 points):** Finding the Leaders

For this question, you are working with SimpleMIPS, an architecture that has a subset of the MIPS architecture instructions. In SimpleMIPS there are only two instructions that effect control flow transfers: **bne** and **j. **bne** is a conditional branch instruction that takes as argument two registers and a 16-bit constant. The branch “falls through” if the values stored in the specified registers are equal. The branch “is taken” if these values are different. When the branch falls through the 4 is added to the current value of the program counter (PC). The 16-bit constant is the displacement that is to be added to the PC. If the branch is taken, first 4 is added to the current value of the PC, and then the displacement value is sign-extended to 32 bits and added to the PC. Thus the operation of the **bne** instruction can be summarized as follows:

\[
bne \; \$1, \; \$2, \; 8 \iff PC \leftarrow PC + 4 \\
\text{if}($1 \neq $2) \; \text{then} \; PC \leftarrow PC + 8
\]

To sign extend a 16-bit constant to a 32-bit one, we have to copy the value of the bit 15 of the 16-bit constant into the upper 16 bits of the 32-bit constant.

The format of the **bne** instruction in SimpleMIPS is as follow:

<table>
<thead>
<tr>
<th>Field</th>
<th># of Bits</th>
<th>Bits Occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>6</td>
<td>31-26</td>
</tr>
<tr>
<td>Source Register</td>
<td>5</td>
<td>25-21</td>
</tr>
<tr>
<td>Target Register</td>
<td>5</td>
<td>20-16</td>
</tr>
<tr>
<td>Displacement</td>
<td>16</td>
<td>15-0</td>
</tr>
</tbody>
</table>

The binary value of the OpCode for **bne** is 000101.

The only parameter of the unconditional jump instruction, **j**, is a 26-bit displacement constant. The **j** instruction takes the 4 most significant bits of the PC and concatenates with the 26-bit displacement constant to form a 30-bit value. This 32 bit value is then left-shifted by two to form the 32-bit value that is written in the PC to effect the control flow.

Thus the operation of the **j** instruction can be summarized as follows:

\[
j \; \text{Exit} \iff PC \leftarrow \text{concat}(PC[31-28],IR[25-0])\ll2
\]

Where \(PC[31-28]\) are the bits 31-28 of the PC, and \(IR[25-0]\) are the bits 25-0 of the instruction register.

The format of the **j** instruction in SimpleMIPS is as follow:

<table>
<thead>
<tr>
<th>Field</th>
<th># of Bits</th>
<th>Bits Occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>6</td>
<td>31-26</td>
</tr>
<tr>
<td>Displacement</td>
<td>26</td>
<td>25-0</td>
</tr>
</tbody>
</table>

The binary value of the OpCode for **j** is 000010.

You got a Summer internship at Reverse Engineering Inc. You are asked to write an algorithm that will be used to build a CFG from binary files containing code for SimpleMIPS. Assume that the following functions are available in the “decompiler” in which you are working:
unsigned int *CreateBitVector(unsigned int VectorLength);
void SetBit(unsigned int *BitVector, unsigned int BitNumber);
void ResetBitRange(unsigned int *BitVector, unsigned int FirstBit,
                  unsigned int LastBit);
void ResetBit(unsigned int *BitVector, unsigned int BitNumber);

CreateBitVector allocates enough memory to store a bit vector of VectorLength bits. SetBit
sets the bit BitNumber to 1, ResetBit sets the bit BitNumber to 0, and ResetBitRange sets the
bit from FirstBit to LastBit, inclusive, to 0.

You are asked to finish writing the code for the function BBLeaders that returns a pointer
to a bit vector where the bit corresponding to instruction i is 1 if and only if i is a basic block
leader. BBLeaders receives as parameters a pointer to the first instruction and a pointer to the last
instruction, inclusive, that has to be inspected.

Another programmer has started writing the BBLeaders C code as follows:

unsigned int *BBLeaders(unsigned int *FirstInstruction, unsigned int *LastInstruction)
{
  unsigned int *instruction;
  unsigned int *bitvector;
  int bitcount = 0;
  unsigned int opcode;
  unsigned int num_instructions;

  if (LastInstruction <= FirstInstruction)
    return 0;
  num_instructions = LastInstruction - FirstInstruction+1;
  /* Need to write here the code to create and initialize the bitvector */

  for(instruction = FirstInstruction;
      instruction <= LastInstruction;
      instruction++)
    {
      /* Loop Body to process each instruction --- should use bitcount
to maintain the position within the bitvector */
    }
  return bitvector;
}

In order to complete BBLeaders, you have to write the code to create and initialize the
bitvector, and the code for the loop body that will process each instruction and set the bits
in the bitvector to their correct values.

You will be evaluated according to the following criteria:

a. (10 points) Correct creation and initialization of the bitvector;

b. (5 points) Correct control structure within the loop;
c. (15 points) Correct processing of each instruction type, and correct generation of the bitvector.

To solve this question, it is useful to recall the following bitwise operators in C:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>~x</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
<td>x &lt;&lt; y</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
<td>x &gt;&gt; y</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>x &amp; y</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td>x ^ y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
</tr>
</tbody>
</table>